18 Highly Expressive Languages

Alice:	*I still cannot check if I have an even number of shoes.*
Riccardo:	*This will not stand!*
Sergio:	*We now provide languages that do just that.*
Vittorio:	*They can also express any query you can think of.*

In previous chapters, we studied a number of powerful query languages, such as the *fixpoint* and *while* queries. Nonetheless, there are queries that these languages cannot express. As pointed out in the introduction to Chapter 14, *fixpoint* lies within PTIME, and *while* within PSPACE. The complexity bound implies that there are queries, of complexity higher than PSPACE, that are not expressible in the languages considered so far. Moreover, we showed simple, specific queries that are not in *fixpoint* or *while*, such as the query *even*.

In this chapter, we exhibit several powerful languages that have no complexity bound on the queries they can express. We build up toward languages that are complete (i.e., they express all queries). Recall that the notion of query was made formal in Chapter 16. Basically, a query is a mapping from instances of a fixed input schema to instances of a fixed answer schema that is computable and generic. Recall that, as a consequence, answers to queries contain only constants from the input (except possibly for some fixed, finite set of new constants).

We begin with a language that extends *while* by providing arbitrary computing power *outside* the database; this yields a language denoted while_N, in the style of embedded relational languages like C+SQL. This would seem to provide the simplest cure for the computational limitations of the languages exhibited so far. There is no complexity bound on the queries while_N can express. Surprisingly, we show that, nonetheless, while_N is not complete. In fact, while_N cannot express certain simple queries, including the infamous query *even*. Intuitively, while_N is not complete because the external computation has limited interaction with the database. Complete languages are obtained by overcoming this limitation. Specifically, we present two ways to do this: (1) by extending *while* with the ability to create new values in the course of the computation, and (2) by extending *while* with an untyped version of relational algebra that allows relations of variable arity.

For conciseness, in this chapter we do not pursue the simultaneous development of languages in the three paradigms—algebraic, logic, and deductive. Instead we choose to focus on the algebraic paradigm. However, analogous languages could be developed in the other paradigms (see Exercise 18.22).

18.1 *While$_N$—while* with Arithmetic

The language *while* is the most powerful of the languages considered so far. We have seen that it lies within PSPACE. Thus it does not have full computing power. Clearly, a complete language must provide such power. In this section, we consider an extension of *while* that does provide full computing power *outside* the database. Nonetheless, we will show that the resulting language is *not* complete; it is important to understand why this is so before considering more exotic ways of augmenting languages.

The extension of *while* that we consider allows us to perform, outside the database, arbitrary computations on the integers. Specifically, the following are added to the *while* language:

(i) integer variables, denoted $i, j, k, \ldots$;

(ii) the integer constant 0 (zero);

(iii) instructions of the form *increment(i), decrement(i)*, where i is an integer variable;

(iv) conditional statements of the form *if* $i = 0$ then s *else* s', where i is an integer variable and s, s' are statements in the language;

(v) loops of the form *while* $i > 0$ *do* s, where i is an integer variable and s a program.

The semantics is straightforward. All integer variables are initialized to zero. The semantics of the *while change* construct is not affected by the integer variables (i.e., the loop is executed as long as there is a change in the content of a *relational* variable). The resulting language is denoted by *while$_N$*.

Because the language *while$_N$* can simulate an arbitrary number of counters, it is computationally complete on the integers (see Chapter 2). More precisely, the following holds:

Fact For every computable function $f(i_1, \ldots, i_k)$ on integers, there exists a *while$_N$* program w_f that computes $f(i_1, \ldots, i_k)$ for every integer initialization of $i_1, \ldots, i_k$. In particular, w_f stops on input $i_1, \ldots, i_k$ iff f is defined on $(i_1, \ldots, i_k)$.

In view of this fact, one can use in *while$_N$* programs, whenever convenient, statements of the form $n := f(i_1, \ldots, i_k)$, where $n, i_1, \ldots, i_k$ are integer variables and f is a computable function on the integers. This is used in the following example.

EXAMPLE 18.1.1 Let G be a binary relation with attributes AB. Consider the query on the graph G:

$$\textit{square}(G) = \emptyset \text{ if the diameter of } G \text{ is a perfect square, and } G \text{ otherwise.}$$

The following *while$_N$* program computes *square*(G) (the output relation is *answer*; it is assumed that $G \neq \emptyset$):

$i := 0; T := G;$

```
while change do
   begin
   $T := T \cup \pi_{AB}(\delta_{B\to C}(T) \bowtie \delta_{A\to C}(G))$;
   increment($i$);
   end;
$j := f(i)$;
$answer := G$;
if $j > 0$ then $answer := \emptyset$.
```

where f is the function such that $f(x) = 1$ if x is a perfect square and $f(x) = 0$ otherwise. (Clearly, f is computable.) Note that, after execution of the while loop, the value of i is the diameter of G.

It turns out that the preceding program can been expressed in *while* alone, and even *fixpoint*, without the need for arithmetic (see Exercise 18.2). However, this is clearly not the case in general. For instance, consider the $while_N$ program obtained by replacing f in the preceding program by some arbitrary computable function.

Despite its considerable power, $while_N$ cannot express certain simple queries, such as *even*. There are several ways to show this, just as we did for *while*. Recall that, in Chapter 17, it was shown that *while* has a 0-1 law. It turns out that $while_N$ also has a 0-1 law, although proving this is beyond the scope of this book. Thus there are many queries, including *even*, that $while_N$ cannot express. One can also give a direct proof that *even* cannot be expressed by $while_N$ by extending straightforwardly the hyperplane technique used in the direct proof that *while* cannot express *even* (Proposition 17.3.2, see Exercise 18.3).

As in the case of other languages we considered, order has a significant impact on the expressiveness of $while_N$. Indeed, $while_N$ is complete on ordered databases.

Theorem 18.1.2 The language $while_N$ expresses all queries on ordered databases.

Crux Let q be a query on an ordered database with schema **R**. Let **I** denote an input instance over **R** and α the enumeration of constants in **I** given by the relation *succ*. By the definition of query, there exists a Turing machine M_q that, given as input $enc_\alpha(\mathbf{I})$, produces as output $enc_\alpha(q(\mathbf{I}))$ (whenever q is defined on **I**). Because $while_N$ manipulates integers, we wish to encode **I** as an integer rather than a Turing machine tape. This can be done easily because each word over some finite alphabet with k symbols (with some arbitrary order among the symbols) can be viewed as an integer in base k. For any instance **J**, let $enc^*_\alpha(\mathbf{J})$ denote the integer encoding of **J** obtained by viewing $enc_\alpha(\mathbf{J})$ as an integer. It is easy to see that there is a computable function f_q on the integers such that $f_q(enc^*_\alpha(\mathbf{I})) = enc^*_\alpha(q(\mathbf{I}))$ whenever q is defined on **I**. Furthermore, because $while_N$ can express any computable function over the integers (see the preceding Fact), there exists a $while_N$ program $w_{f_q}(i)$ that computes f_q. It is left to show that $while_N$ can compute $enc^*_\alpha(\mathbf{I})$ and can decode $q(\mathbf{I})$ from $enc^*_\alpha(q(\mathbf{I}))$. Recall that, in the proof of Theorem 17.4.2, it was shown that *while* can compute a relational representation of $enc_\alpha(\mathbf{I})$ and, conversely, it can decode $q(\mathbf{I})$ from the representation of $enc_\alpha(q(\mathbf{I}))$. A slight modification of that construction can be used to

S

a	b
a	c
c	a

R

a	b	α
a	c	β
c	a	γ

Figure 18.1: An application of *new*

show that $while_N$ can compute the desired integer encoding and decoding. Thus a $while_N$ program computes q in three phases:

1. compute $enc^*_\alpha(\mathbf{I})$;
2. compute $f_q(enc^*_\alpha(\mathbf{I})) = enc^*_\alpha(q(\mathbf{I}))$;
3. compute $q(\mathbf{I})$ from $enc^*_\alpha(q(\mathbf{I}))$. ■

18.2 *While$_{new}$—while* with New Values

Recall that, as discussed in the introduction to Chapter 14, *while* cannot go beyond PSPACE because (1) throughout the computation it uses only values from the input, and (2) it uses relations of fixed arity. The addition of integers as in $while_N$ is one way to break the space barrier. Another is to relax (1) or (2). Relaxing (1) is done by allowing the creation of new values not present in the input. Relaxing (2) yields an extension of *while* with *untyped algebra* (i.e., an algebra of relations with variable arities). In this and the next section, we describe two languages obtained by relaxing (1) and (2) and prove their completeness.

We first present the extension of *while* denoted $while_{new}$, which allows the creation of new values throughout the computation. The language *while* is modified as follows:

(i) There is a new instruction $R := new(S)$, where R and S are relational variables and $arity(R) = arity(S) + 1$;

(ii) The looping construct is of the form *while R do s*, where R is a relational variable.

The semantics of (i) is as follows: Relation R is obtained by extending each tuple of S by one distinct new value from **dom** not occurring in the input, the current state, or in the program. For example, if the value of S is the relation in Fig. 18.1, then R is of the form shown in that figure. The values α, β, γ are distinct new values[1] in **dom**.

The semantics of *while R do s* is that statement s is executed while R is nonempty. We could have used *while change* instead because each looping construct can simulate the other. However, in our context of value invention, it is practical to have the more direct control on loops provided by *while R*.

[1] If $arity(S) = 0$, then R is unary and contains one new value if $S = \{\langle\rangle\}$ and is empty if $S = \emptyset$. This allows the creation of values one by one. One might wonder if this kind of one-by-one value creation is sufficient. The answer is negative. The language with one-by-one value creation is equivalent to $while_N$ (see Exercise 18.6).

Note that the *new* construct is, strictly speaking, nondeterministic. The new values are arbitrary, so several possible outcomes are possible depending on the choice of values. However, the different outcomes differ *only* in the choice of new values. This is formalized by the following:

Lemma 18.2.1 Let w be a $\textit{while}_{\textit{new}}$ program with input schema $\mathbf{R}$, and let R be a relation variable in w. Let $\mathbf{I}$ be an instance over $\mathbf{R}$, and let J, J' be two possible values of R at the same point during the execution of w on $\mathbf{I}$. Then there exists an isomorphism ρ from J to J' that is the identity on the constants occurring in $\mathbf{I}$ or w.

The proof of Lemma 18.2.1 is done by a straightforward induction on the number of steps in a partial execution of w on $\mathbf{I}$ (Exercise 18.7).

Recall that our definition of *query* requires that the answer be unique (i.e., the query must be deterministic). Therefore we must consider only $\textit{while}_{\textit{new}}$ programs whose answer never contains values introduced by the *new* statements. Such programs are called *well-behaved* $\textit{while}_{\textit{new}}$ programs. It is possible to give a syntactic restriction on $\textit{while}_{\textit{new}}$ programs that guarantees good behavior, can be checked, and yields a class of programs equivalent to all well-behaved $\textit{while}_{\textit{new}}$ programs (see Exercises 18.8 and 18.9).

We wish to show that well-behaved $\textit{while}_{\textit{new}}$ programs can express all queries. First we have to make sure that well-behaved $\textit{while}_{\textit{new}}$ programs do in fact express queries. This is shown next.

Lemma 18.2.2 Each well-behaved $\textit{while}_{\textit{new}}$ program with input schema $\mathbf{R}$ and output schema *answer* expresses a query from *inst*($\mathbf{R}$) to *inst(answer)*.

Proof We need to show that well-behaved $\textit{while}_{\textit{new}}$ programs define *mappings* from *inst*($\mathbf{R}$) to *inst(answer)* (i.e., they are deterministic with respect to the final answer). Computability and genericity are straightforward. Let w be a well-behaved $\textit{while}_{\textit{new}}$ program with input schema $\mathbf{R}$ and output *answer*. Let I, I' be two possible values of *answer* after the execution of w on an instance $\mathbf{I}$ of $\mathbf{R}$. By Lemma 18.2.1, there exists an isomorphism ρ from I to I' that is the identity on values in $\mathbf{I}$ or w. Because w is well behaved, *answer* contains *only* values from $\mathbf{I}$ or w. Thus ρ is the identity and $I = I'$. ■

Note that although well-behaved programs are deterministic with respect to their final answer, they are not deterministic with respect to intermediate results that may contain new values.

We next show that well-behaved $\textit{while}_{\textit{new}}$ programs express all queries. The basic idea is simple. Recall that $\textit{while}_N$ is complete on *ordered* databases. That is, for each query q, there is a $\textit{while}_N$ program w that, given an enumeration of the input values in a relation *succ*, computes q. If, given an input, we were able to construct such an enumeration, we could then simulate $\textit{while}_N$ to compute any desired query. Because of genericity, we cannot hope to construct *one* such enumeration. However, constructing *all* enumerations of values in the input would not violate genericity. Both $\textit{while}_{\textit{new}}$ and the language with variable arities considered in the next section can compute arbitrary queries precisely in this fashion: They first compute all possible enumerations of the input values and then simulate a *while*$_N$ program on the ordered database corresponding to each enumeration. These computations yield the same result for all enumerations because queries are generic, so the result is independent of the particular enumeration used to encode the database (see Chapter 16).

Before proving the result, we show how we can construct all the possible enumerations of the elements in the active domain of the input.

Representation

Let **I** be an instance over **R**. Let *Success* be the set of all binary relations defining a successor relation over *adom*(**I**). We can represent all the enumerations in *Success* with a 3-ary relation:

$$\overline{succ} = \bigcup_{I \in Success} I \times \{\alpha_I\},$$

where $\{\alpha_I \mid I \in Success\}$ is a set of distinct new values. [Each such α_I is used to denote a particular enumeration of *adom*(**I**).] For example, Fig. 18.2 represents an instance **I** and the corresponding $\overline{succ}$.

Computation of $\overline{succ}$

We now argue that there exists a *while*$_{new}$ program w that, given **I**, computes $\overline{succ}$. Clearly, there is a *while*$_{new}$ program that, given **I**, produces a unary relation D containing all values in **I**. Following is a *while*$_{new}$ program $w_{\overline{succ}}$ that computes the relation $\overline{succ}$ starting from D (using a query q explained next):

I	
a	*b*
a	*c*
c	*a*

$\overline{succ}$		
a	*b*	α_1
b	*c*	α_1
a	*c*	α_2
c	*b*	α_2
b	*a*	α_3
a	*c*	α_3
b	*c*	α_4
c	*a*	α_4
c	*a*	α_5
a	*b*	α_5
c	*b*	α_6
b	*a*	α_6

$\widehat{succ}$				
a	*b*	*a*	*b*	*c*
b	*c*	*a*	*b*	*c*
a	*c*	*a*	*c*	*b*
c	*b*	a	*c*	*b*
b	*a*	*b*	*a*	*c*
a	*c*	*b*	*a*	*c*
b	*c*	*b*	*c*	*a*
c	*a*	*b*	*c*	*a*
c	*a*	*c*	*a*	*b*
a	*b*	*c*	*a*	*b*
c	*b*	*c*	*b*	*a*
b	*a*	*c*	*b*	*a*

Figure 18.2: An example of $\overline{succ}$ and $\widehat{succ}$

```
$\overline{succ}$ $:= new(\sigma_{1\neq 2}(D \times D))$;
$\Delta$ $:= q$;
while $\Delta$ do
        begin
        $S := new(\Delta)$;
        $\overline{succ} := \left\{ \langle x, y, \alpha' \rangle \;\middle|\; \begin{array}{c} \exists \alpha, x', y'[S(x', y', \alpha, \alpha') \wedge \overline{succ}(x, y, \alpha)] \\ \vee \;\; \exists \alpha[S(x, y, \alpha, \alpha')] \end{array} \right\}$;
        $\Delta := q$;
        end
```

The intuition is that we construct in turn enumerations of subsets of size 2, 3, etc., until we obtain the enumerations of D. (To simplify, we assume that D contains more than two elements.) An enumeration of a subset of D consists of a successor (binary) relation over that subset. As mentioned earlier, the program associates a marking (invented value) with each such successor relation.

During the computation, $\overline{succ}$ contains the successor relation of subsets of size i computed so far. A triple $\langle a, b, \alpha \rangle$ indicates that b follows a in enumeration denoted α.

The first instruction computes the enumerations of subsets of size 2 (i.e., the distinct pairs of elements of D) and marks them with new values. At each iteration, Δ indicates for each enumeration the elements that are missing in this enumeration. More precisely, relation Δ must contain the following set of triples:

$$\left\{ \langle a, b, \alpha \rangle \;\middle|\; \begin{array}{r} b \text{ does not occur in the successor relation corresponding to } \alpha \\ \text{and the last element of } \alpha \text{ is } a. \end{array} \right\}$$

The relational query q computes the set Δ given a particular relation $\overline{succ}$. If Δ is not empty, for each α a new value α' is created for each element missing in α (i.e., the enumeration α is extended in all possible ways with each of the missing elements). This yields as many new enumerations from each α as missing elements.

This is iterated until Δ becomes empty, at which point all enumerations are complete. Note that if D contains n elements, the final result $\overline{succ}$ contains $n!$ enumerations.

Theorem 18.2.3 The well-behaved $while_{new}$ programs express all queries.

Crux Let q be a query from $inst(\mathbf{R})$ to $inst(answer)$. Assume the query is generic (i.e., C-generic with $C = \emptyset$). The proof is easily modified for the case when the query is C-generic with $C \neq \emptyset$. It is sufficient to observe that

(*) for each $while_N$ program,
there exists an equivalent well-behaved $while_{new}$ program.

Suppose that (*) holds. Let $w_{\overline{succ}}$ be the $while_{new}$ program computing $\overline{succ}$ from given $\mathbf{I}$ over $\mathbf{R}$. By Theorem 18.1.2 and (*), there exists a $while_{new}$ program $w(succ)$ that computes q using a successor relation $succ$. We construct another $while_{new}$ program $\overline{w}(\overline{succ})$ that computes q given $\mathbf{I}$ and $\overline{succ}$. Intuitively, $w(succ)$ is run in parallel for *all* possible enumerations *succ* provided by $\overline{succ}$. All computations produce the same result and are placed in *answer*. The computations for different enumerations in $\overline{succ}$ are identified by the α marking the enumeration in $\overline{succ}$. To this end, each relation R of arity k in $w(succ)$ is replaced by a relation $\overline{R}$ of arity $k+1$. The extended database relations are first initialized by statements of the form $\overline{R} := R \times \pi_3(\overline{succ})$. Next the instructions of $w(succ)$ are modified as follows:

- $R := \{\langle u\rangle \mid \phi(u)\}$ becomes $\overline{R} := \{\langle u, \alpha\rangle \mid \exists y \exists z \overline{succ}(y, z, \alpha) \wedge \overline{\phi}(u, \alpha)\}$, where $\overline{\phi}(u, \alpha)$ is obtained from $\phi(u)$ by replacing each atom $S(v)$ by $\overline{S}(v, \alpha)$;
- *while change do* remains unchanged.

Finally the instruction *answer* $:= \pi_{1..n}(\overline{answer})$, where $n = arity(answer)$, is appended at the end of the program. The following can be shown by induction on the steps of a partial execution of $\overline{w}(\overline{succ})$ on **I** (Exercise 18.10):

(**) At each point in the computation of $\overline{w}(\overline{succ})$ on **I**, the set of tuples in relation $\overline{R}$ marked with α coincides with the value of R at the same point in the computation when $w(succ)$ is run on **I** and *succ* is the successor relation corresponding to α.

In particular, at the end of the computation of $\overline{w}(\overline{succ})$ on **I**,

$$\overline{answer} = \bigcup_{\alpha} w(\alpha)(\mathbf{I}) \times \{\alpha\},$$

where α ranges over the enumeration markers. Because $w(\alpha)(\mathbf{I}) = q(\mathbf{I})$ for each α, it follows that *answer* contains $q(\mathbf{I})$ at the end of the computation. Thus query q is computable by a well-behaved *while$_{new}$* program.

Thus it remains to show (*). Integer variables are easily simulated as follows. An integer variable i is represented by a binary variable R_i. If i contains the integer n, then R_i contains a successor relation for $n+1$ distinct new values:

$$\{\langle \alpha_j, \alpha_{j+1}\rangle \mid 0 \leq j < n\}.$$

(The integer 0 is represented by an empty relation and the integer 1 by a singleton $\{\langle \alpha_0, \alpha_1\rangle\}$.) It is easy to find a *while$_{new}$* program for *increment* and *decrement* of i. ■

We showed that well-behaved *while$_{new}$* programs are complete with respect to our definition of query. Recall that *while$_{new}$* programs that are not well behaved can compute a different kind of query that we excluded deliberately, which contains new values in the answer. It turns out, however, that such queries arise naturally in the context of object-oriented databases, where new object identifiers appear in query results (see Chapter 21). This requires extending our definition of query. In particular, the query is nondeterministic but, as discussed earlier, the different answers differ only in the particular choice of new values. This leads to the following extended notion of query:

Definition 18.2.4 A *determinate query* is a relation Q from *inst*(**R**) to *inst(answer)* such that

$$\begin{tikzcd}
 & & \psi_3 \arrow[dl] \arrow[drr] & & \\
\{a, b\} \Rightarrow & \psi_0 \arrow[r] \arrow[dr] & b & \psi_2 \arrow[l] \arrow[ul] & a \\
 & & \psi_1 \arrow[ur] \arrow[urr] & &
\end{tikzcd}$$

Figure 18.3: A query not expressible in $while_{new}$

- Q is computable;
- if $\langle I, J\rangle \in Q$ and ρ is a one-to-one mapping on constants, then $\langle \rho(I), \rho(J)\rangle \in Q$; and
- if $\langle I, J\rangle \in Q$ and $\langle I, J'\rangle \in Q$, then there exists an isomorphism from J to J' that is the identity on the constants in I.

A language is *determinate complete* if it expresses only determinate queries and all determinate queries.

Let Q be a determinate query. If $\langle I, J\rangle \in Q$ and ρ is a one-to-one mapping on constants leaving I fixed, then $\langle I, \rho(J)\rangle \in Q$.

The question arises whether $while_{new}$ remains complete with respect to this extended notion of query. Surprisingly, the answer is negative. Each $while_{new}$ query is determinate. However, we exhibit a simple determinate query that $while_{new}$ cannot express. Let q be the query with input schema $\mathbf{R} = \{S\}$, where S is unary, and output G, where G is binary. Let q be defined as follows: For each input I over S, if $I = \{a, b\}$ then $q(I) = \{\langle\psi_0, \psi_1\rangle, \langle\psi_1, \psi_2\rangle, \langle\psi_2, \psi_3\rangle, \langle\psi_3, \psi_0\rangle, \langle\psi_0, b\rangle, \langle\psi_1, a\rangle, \langle\psi_2, b\rangle, \langle\psi_3, a\rangle\}$ for some new elements $\psi_0, \psi_1, \psi_2, \psi_3$, and $q(I) = \emptyset$ otherwise (Fig. 18.3).

Theorem 18.2.5 The query q is not expressible in $while_{new}$.

Proof The proof is by contradiction. Suppose w is a $while_{new}$ program expressing q. Consider the sequence of steps in the execution of w on an input $I = \{a, b\}$. We can assume without loss of generality that no invented value is ever deleted from the database (otherwise modify the program to keep all invented values in some new unary relation). For each invented value occurring in the computation, we define a trace that records how the value was invented and uniquely identifies it. More precisely, $trace(\alpha)$ is defined inductively as follows. If α is a constant, then $trace(\alpha) = \langle\alpha\rangle$. Suppose α is a new value created at step i with a *new* statement associating it with tuple $\langle x_1, \ldots, x_k\rangle$. Then $trace(\alpha) = \langle i, trace(x_1), \ldots, trace(x_k)\rangle$. Clearly, one can extend *trace* to tuples and relations in the natural manner. It is easily shown (Exercise 18.11) by induction on the number of steps in a partial execution of w on I that

(†) $trace(\alpha) = trace(\beta)$ iff $\alpha = \beta$;

(‡) for each instance J computed during the execution of w on input I, $trace(J)$ is closed under each automorphism ρ of I. In particular, for each α occurring in J, $\rho(trace(\alpha))$ equals $trace(\beta)$ for some β also occurring in J.

Consider now $trace(q(I))$ and the automorphism ρ of I [and therefore of $trace(q(I))$] defined by $\rho(a) = b$, $\rho(b) = a$. Note that $\rho^2 = id$ (the identity) and $\rho = \rho^{-1}$. Consider $\rho(trace(\psi_0))$. Because $\langle \psi_0, b\rangle \in q(I)$, it follows that $\langle trace(\psi_0), b\rangle \in trace(q(I))$. Because $\rho(b) = a$, it further follows that $\langle \rho(trace(\psi_0)), a\rangle \in trace(q(I))$ so $\rho(trace(\psi_0))$ is either $trace(\psi_1)$ or $trace(\psi_3)$. Suppose $\rho(trace(\psi_0)) = trace(\psi_1)$ (the other case is similar). From the fact that ρ is an automorphism of $trace(q(I))$ it follows that $\rho(trace(\psi_3)) = trace(\psi_0)$, $\rho(trace(\psi_2)) = trace(\psi_3)$, and $\rho(trace(\psi_1)) = trace(\psi_2)$. Consider now ρ^2. First, because $\rho^2 = id$, $\rho^2(trace(\psi_i)) = trace(\psi_i), 0 \le i \le 3$. On the other hand, $\rho^2(trace(\psi_0)) = \rho(\rho(trace(\psi_0))) = \rho(trace(\psi_1)) = trace(\psi_2)$. This is a contradiction. Hence q cannot be computed by *while*$_{new}$. ■

The preceding example shows that the presence of new values in the answer raises interesting questions with regard to completeness. There exist languages that express all queries with invented values in answers (see Exercise 18.14 for a complex construct that leads to a determinate-complete language). Value invention is common in object-oriented languages, in the form of object creation constructs (see Chapter 21).

18.3 *While*$_{uty}$—An Untyped Extension of *while*

We briefly describe in this section an alternative complete language obtained by relaxing the fixed-arity requirement of the languages encountered so far. This relaxation is done using an *untyped* version of relational algebra instead of the familiar typed version. We will obtain a language allowing us to construct relations of variable, data-dependent arity in the course of the computation. Although strictly speaking they are not needed, we also allow integer variables and integer manipulation, as in *while*$_N$. Intuitively, it is easy to see why this yields a complete language. Variable arities allow us to construct all enumerations of constants in the input, represented by sufficiently long tuples containing all constants. The ability to construct the enumerations and manipulate integers yields a complete language.

The first step in defining the untyped version of *while* is to define an untyped version of relational algebra. This means that operations must be defined so that they work on relations of arbitrary, unknown arity. Expressions in the untyped algebra are built from relation variables and constants and can also use *integer* variables and constants. Let i, j be integer variables, and for each integer k, let $\emptyset^k$ denote the empty relation of arity k. Untyped algebra expressions are built up using the following operations:

- If e, e' are expressions, then $e \cap e'$ and $e \cup e'$ are expressions; if $arity(e) = arity(e')$ the semantics is the usual; otherwise the result is $\emptyset^0$.

- If e is an expression, then $\neg e$ is an expression; the complement is with respect to the active domain (not including the integers).
- If e, f are expressions, then $e \times f$ is an expression; the semantics is the usual cross-product semantics.
- If e is an expression, then $\sigma_{i=j}(e)$ is an expression, where i, j are integer variables or constants; if $arity(e) \geq max\{i, j\}$ the semantics is the usual; otherwise the result is $\emptyset^0$.
- If e is an expression, then $\pi_{ij}(e)$ is an expression, where i, j are integer variables or constants; if $i \leq j$ and $arity(e) \geq max\{i, j\}$, this projects e on columns i through j; otherwise the result is $\emptyset^{|j-i|}$.
- If e is an expression, then $ex_{ij}(e)$ is an expression; if $arity(e) \geq max\{i, j\}$, this exchanges in each tuple in the result of e the i and j coordinates; otherwise the result is $\emptyset^0$.

We may also consider an untyped version of tuple relational calculus (see Exercise 18.15).

We can now define $while_{uty}$ programs. They are concatenations of statements of the form

- $i := j$, where i is an integer variable and j an integer variable or constant.
- $increment(i)$, $decrement(i)$, where i is an integer variable.
- *while* $i > 0$ *do* t, where i is an integer variable and t a program.
- $R := e$, where R is a relational variable and e an untyped algebra expression; the semantics here is that R is assigned the content *and arity* of e.
- *while* R *do* t, where R is a relational variable and t a program; the semantics is that the body of the loop is repeated as long as R is nonempty.

All relational variables that are not database relations are initialized to $\emptyset^0$; integer variables are initialized to 0.

EXAMPLE 18.3.1 Following is a $while_{uty}$ program that computes the arity of a nonempty relation R in the integer variable n:

```
$S_0 := \{\langle\rangle\}$; $S_1 := S_0 \cup R$; $S_2 := \neg S_1$;
while $S_2$ do
  begin
  $n := n + 1$;
  $S_0 := S_0 \times D$;
  $S_1 := S_0 \cup R$;
  $S_2 := \neg S_1$;
  end
```

where D abbreviates an algebra expression computing the active domain [e.g., $\pi_{11}(R) \cup \neg\pi_{11}(R)$]. The program tries out increasing arities for R starting from 0. Recall that whenever R and S_0 have different arities, the result of $S_0 \cup R$ is $\emptyset^0$. This allows us to detect when the appropriate arity has been found.

Remark 18.3.2 There is a much simpler set of constructs that yields the same power as *while*$_{uty}$. In general, programs are much harder to write in the resulting language, called QL, than in *while*$_{uty}$. One can show that the set of constructs of QL is minimal. The language QL is described next; it does not use integer variables. QL expressions are built from relational variables and constant relations as follows (D denotes the active domain):

- *equal* is an expression denoting $\{\langle a, a\rangle \mid a \in D\}$.
- $e \cap e'$ and $\neg e$ are defined as for *while*$_{uty}$; the complement is with respect to the active domain.
- If e is an expression, then $e \downarrow$ is an expression; this projects out the last coordinate of the result of e (and is $\emptyset^0$ if the arity is already zero).
- If e is an expression, then $e \uparrow$ is an expression; this produces the cross-product of e with D.
- If e is an expression, then $e \sim$ is an expression; if $arity(e) \geq 2$, then this exchanges the last two coordinates in each tuple in the result of e. Otherwise the answer is $\emptyset^0$.

Programs are built by concatenations of assignment statements ($R := e$) and *while* statements (*while R do s*). The semantics of the *while* is that the loop is iterated as long as R is nonempty.

We leave it to the reader to check that QL is equivalent to *while*$_{uty}$ (Exercise 18.17). We briefly describe the simulation of integers by QL. Let Z denote the constant 0-ary relation $\{\langle\rangle\}$. We can have Z represent the integer 0 and $Z \uparrow^n$ represent the integer n. Then *increment*(n) is simulated by one application of $\uparrow$, and *decrement*(n) is simulated by one application of $\downarrow$. A test of the form $x = 0$ becomes $e \downarrow = \emptyset$, where e is the untyped algebra expression representing the value of x. Thus we can simulate arbitrary computations on the integers.

Recall that our definition of query requires that both the input and output be instances over *fixed* schemas. On the other hand, in *while*$_{uty}$ relation arities are variable, so in general the arity of the answer is data dependent. This is a problem analogous to the one we encountered with *while*$_{new}$, which generally produces new values in the result. As in the case of *while*$_{new}$, we can define semantic and syntactic restrictions on *while*$_{uty}$ programs that guarantee that the programs compute queries. Call a *while*$_{uty}$ program *well behaved* if its answer is always of the same arity regardless of the input. Unfortunately, it can be shown that it is undecidable if a *while*$_{uty}$ program is well behaved (Exercise 18.19). However, there is a simple syntactic condition that guarantees good behavior and covers all well-behaved programs. A *while*$_{uty}$ program with answer relation *answer* is *syntactically well behaved* if the last instruction of the program is of the form *answer* $:= \pi_{mn}(R)$, where m, n are integer constants. Clearly, syntactic good behavior guarantees good behavior and can be checked. Furthermore, it is obvious that each well-behaved *while*$_{uty}$ program is equivalent to some syntactically well-behaved program (Exercise 18.19).

We now prove the completeness of well-behaved $while_{uty}$ programs.

Theorem 18.3.3 The well-behaved $while_{uty}$ programs express all queries.

Crux It is easily verified that all well-behaved $while_{uty}$ programs define queries. The proof that every query can be expressed by a well-behaved $while_{uty}$ program is similar to the proof of Theorem 18.2.3. Let q be a query with input schema $\mathbf{R}$. We proceed in two steps: First construct all orderings of constants from the input. Next simulate the $while_N$ program computing q on the ordered database corresponding to each ordering. The main difference with $while_{new}$ lies in how the orderings are computed. In $while_{uty}$, we use the arbitrary arity to construct a relation $R_<$ containing sufficiently long tuples each of which provides an enumeration of all constants. This is done by the following $while_{uty}$ program, where D stands for an algebra expression computing the active domain:

```
$R_< := \emptyset^0$;
$C := D$; $arityC := 1$;
$while\ C\ do$
    $begin$
    $R_< := C$;
    $C := C \times D$; $increment(arityC)$;
    $for\ i := 1\ to\ (arityC - 1)\ do$
        $C := C \cap \neg\sigma_{i=arity(C)}(C)$;
    $end$
```

Clearly, the looping construct $for\ i := 1\ to\ \ldots$ can be easily simulated. If the size of D is n, the result of the program is the set of n-tuples with distinct entries in $adom(D)$. Note that each such tuple t in $R_<$ provides a complete enumeration of the constants in D. Next one can easily construct a $while_{uty}$ program that constructs, for each such tuple t in $R_<$, the corresponding successor relation. More precisely, one can construct

$$\widehat{succ} = \bigcup_{t \in R_<} succ_t \times \{t\},$$

where $succ_t = \{\langle t(i), t(i+1)\rangle \mid 1 \leq i < n\}$ (see Fig. 18.2 and Exercise 18.20). ■

Untyped languages allow us to relax the restriction that the output schema is fixed. This may have a practical advantage because in some applications it may be necessary to have the output schema depend on the input data. However, in such cases one would likely prefer a richer type system rather than no typing at all.

The overall results on the expressiveness and complexity of relational query languages are summarized in Figs. 18.4 and 18.5. The main classes of queries and their inclusion structure are represented in Fig. 18.4 (solid arrows indicate strict inclusion; the dotted arrow indicates strict inclusion if PTIME $\neq$ PSPACE). Languages expressing each class of queries are listed in Fig. 18.5, which also contains information on complexity (first without assumptions, then with the assumption of an order on the database). In Fig. 18.5,

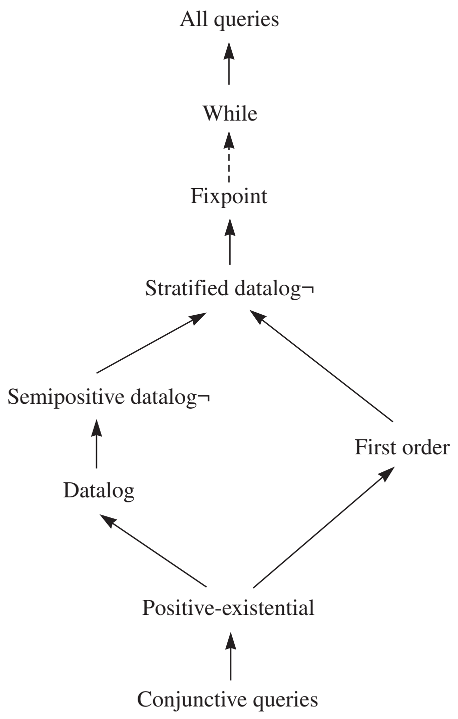

Figure 18.4: Main classes of queries

CALC($\exists$, $\wedge$) denotes the conjunctive calculus and CALC($\exists$, $\wedge$, $\vee$) denotes the positive-existential calculus.

Bibliographic Notes

The first complete language proposed was the language QL of Chandra and Harel [CH80b]. Chandra also considered a language equivalent to $while_N$, which he called LC [Cha81a]. It was shown that LC cannot compute *even*. Several other primitives are considered in [Cha81a] and their power is characterized. The language $while_{new}$ was defined in [AV90], where its completeness was also shown.

The languages considered in this chapter can be viewed as formalizing practical languages, such as C+SQL or O_2C, used to develop database applications. These languages combine standard computation (C) with database computation (SQL in the relational world or O_2 in the object-oriented world). In this direction, several computing devices were defined in [AV91b], and complexity-theoretic results are obtained using the devices. First an extension of Turing machines with a relational store, called *relational machine*, was shown to be equivalent to $while_N$. A further extension of relational machines equivalent to $while_{new}$ and $while_{uty}$, called *generic machine*, was also defined. In the generic machine,

Class of queries	*Languages*	*Complexity*	*Complexity* with order
conjunctive	CALC($\exists, \wedge$)		
	SPJR algebra	$\subset$ LOGSPACE $\subset$ AC$_0$	$\subset$ LOGSPACE $\subset$ AC$_0$
positive-existential	CALC($\exists, \wedge, \vee$)		
	SPJUR algebra		
	nr-datalog	$\subset$ LOGSPACE $\subset$ AC$_0$	$\subset$ LOGSPACE $\subset$ AC$_0$
datalog	datalog	$\subset$ monotonic PTIME	$\subset$ monotonic PTIME
semipositive datalog$^\neg$	semipositive datalog$^\neg$	$\subset$ PTIME	$=$ PTIME (with *min*, *max*)
first order	CALC		
	ALG		
	nr-stratified datalog$^\neg$	$\subset$ LOGSPACE $\subset$ AC$_0$	$\subset$ LOGSPACE $\subset$ AC$_0$
stratified datalog$^\neg$	stratified datalog$^\neg$	$\subset$ PTIME	$=$ PTIME
fixpoint	CALC+μ^+		
	while$^+$		
	datalog$^\neg$ (fixpoint and well-founded semantics)	$\subset$ PTIME	$=$ PTIME
while	CALC+μ		
	while		
	datalog$^{\neg\neg}$ (fixpoint semantics)	$\subset$ PSPACE	$=$ PSPACE
all queries	*while*$_{uty}$	no bound	no bound
	while$_{new}$		

Figure 18.5: Languages and complexity

parallelism is used to allow simultaneous computations with all possible successor relations.

Queries with new values in their answers were first considered in [AK89], in the context of an object-oriented deductive language with object creation, called IQL. The notion of determinate query [VandBGAG92] is a recasting of the essentially equivalent notion of db transformation, formulated in [AK89]. In [AK89], the query in Theorem 18.2.5 is also exhibited, and it is shown that IQL without duplicate elimination cannot express it. Because IQL is more powerful than *while*$_{new}$, their result implies the result of Theorem 18.2.5. The issue of completeness of languages with object creation was further investigated in [AP92, VandBG92, VandBGAG92, VandBP95, DV91, DV93].

Finally it is easy to see that each (determinate) query can be computed in some natural *nondeterministic* extension of $while_{new}$ (e.g., with the witness operator of Chapter 17) [AV91c]. However, such programs may be nondeterministic so they do not define only determinate queries.

Exercises

Exercise 18.1 Let G be a graph. Consider a query "*Does the shortest path from a to b in G have property P?*" where G is a graph, P is a recursive property of the integers, and a, b are two particular vertexes of the graph. Show that such a query can be expressed in $while_N$.

Exercise 18.2 Prove that the query in Example 18.1.1 can be expressed (a) in *while*; (b) in *fixpoint*.

Exercise 18.3 Sketch a direct proof that *even* cannot be expressed by $while_N$ by extending the hyperplane technique used in the proof of Proposition 17.3.2.

♠ **Exercise 18.4** [AV94] Consider the language $\mathcal{L}$ augmenting $while_N$ by allowing mixing of integers with data. Specifically, the following instruction is allowed in addition to those of $while_N$: $R := \{\langle i_1, \ldots, i_k \rangle\}$, where R is a k-ary relation variable and $i_1, \ldots, i_k$ are integer variables. It is assumed that the domain of input values is disjoint from the integers. Complement (or negation) is taken with respect to the domain formed by all values in the database or program, including the integer values present in the database. The *well-behaved* $\mathcal{L}$ programs are those whose outputs never contain integers. Show that well-behaved $\mathcal{L}$ and $while_N$ are equivalent.

Exercise 18.5 Complete the proof of Theorem 18.1.2.

♠ **Exercise 18.6** [AV90] Consider a variation of the language $while_{new}$ where the $R := new(S)$ instruction is replaced by the simpler instruction "$R := new$" where R is unary. The semantics of this instruction is that R is assigned a singleton $\{\langle \alpha \rangle\}$, where α is a new value. Denote the new language by $while_{unary\text{-}new}$.

(a) Show that each query expressible in $while_N$ is also expressible in $while_{unary\text{-}new}$.
Hint: Use new values to represent integers. Specifically, to represent the integers up to n, construct a relation $succ_{int}$ containing a successor relation on n new values. The value of rank i with respect to *succ* represents integer i.

(b) Show that each query expressible in $while_{unary\text{-}new}$ is also expressible in $while_N$.
Hint: Again establish a correspondence between new values and integers. Then use Exercise 18.4.

Exercise 18.7 Prove Lemma 18.2.1.

Exercise 18.8 Prove that it is undecidable if a given $while_{new}$ program is well behaved.

★ **Exercise 18.9** In this exercise we define a syntactic restriction on $while_{new}$ programs that guarantees good behavior. Let w be a $while_{new}$ program. Without loss of generality, we can assume that all instructions contain at most one algebraic operation among $\cup, -, \pi, \times, \sigma$. Let the *not-well-behaved set of* w, denoted $Bad(w)$, be the smallest set of pairs of the form $\langle R, i \rangle$, where R is a relation in w and $1 \leq i \leq arity(R)$, such that

(a) if $S := new(R)$ is an instruction in w and $arity(S) = k$, then $\langle S, k\rangle \in Bad(w)$;

(b) if $S := T \cup R$ is in w and $\langle T, i\rangle \in Bad(w)$ or $\langle R, i\rangle \in Bad(w)$, then $\langle S, i\rangle \in Bad(w)$;

(c) if $S := T - R$ is in w and $\langle T, i\rangle \in Bad(w)$, then $\langle S, i\rangle \in Bad(w)$;

(d) if $S := T \times R$ is in w and $\langle T, i\rangle \in Bad(w)$, then $\langle S, i\rangle \in Bad(w)$; and if $\langle R, j\rangle \in Bad(w)$, then $\langle S, arity(T) + j\rangle \in Bad(w)$;

(e) if $S := \pi_{i_1 \ldots i_k}(T)$ is in w and $\langle T, i_j\rangle \in Bad(w)$, then $\langle S, j\rangle \in Bad(w)$;

(f) if $S := \sigma_{cond}(T)$ is in w and $\langle T, i\rangle \in Bad(w)$, then $\langle S, i\rangle \in Bad(w)$.

A $while_{new}$ program w is *syntactically well behaved* if

$$\{\langle answer, i\rangle \mid 1 \leq i \leq arity(answer)\} \cap Bad(w) = \emptyset.$$

(a) Outline a procedure to check that a given $while_{new}$ program is syntactically well behaved.

(b) Show that each syntactically well-behaved $while_{new}$ program is well behaved.

(c) Show that for each well-behaved $while_{new}$ program, there exists an equivalent syntactically well-behaved $while_{new}$ program.

Exercise 18.10 Prove (*) in the proof of Theorem 18.2.3.

Exercise 18.11 Prove (†) and (‡) in the proof of Theorem 18.2.5.

Exercise 18.12 Consider the query q exhibited in the proof of Theorem 18.2.5. Let q_2 be the query that, on input $I = \{a, b\}$, produces as answer two copies of $q(I)$. More precisely, for each ψ_i in $q(I)$, let ψ_i' be a distinct new value. Let $q'(I)$ be obtained from $q(I)$ by replacing ψ_i by ψ_i', and let $q_2(I) = q(I) \cup q'(I)$. Prove that q_2 can be expressed by a $while_{new}$ program.

♠ **Exercise 18.13** [DV91, DV93] Consider the instances I, J of Fig. 18.6. Consider a query q that, on input of the same pattern as I, returns J (up to an arbitrary choice of distinct β, θ_i) and otherwise returns the empty instance. Show that q is not expressible in $while_{new}$.

♠ **Exercise 18.14** (*Choose* [AK89]) Let $while_{new}^{choose}$ be obtained by augmenting $while_{new}$ with the following (determinate) *choose* construct. A program w may contain the instruction $choose(R)$ for some unary relation R. On input $\mathbf{I}$, when $choose(R)$ is applied in a state $\mathbf{J}$, the next state $\mathbf{J}'$ is defined as follows:

(a) if for each a, b in $\mathbf{J}(R)$, there is an automorphism of $\mathbf{J}$ that is the identity over $adom(\mathbf{I}, w)$ and maps a to b, $\mathbf{J}'$ is obtained from $\mathbf{J}$ by eliminating one arbitrary element in $\mathbf{J}(R)$;

(b) otherwise $\mathbf{J}'$ is just $\mathbf{J}$.

Show that $while_{new}^{choose}$ is determinate complete.

Exercise 18.15 One may consider an untyped version of tuple relational calculus. Untyped relations are used just like typed relations, except that terms of the form $t(i)$ are allowed, where t is a tuple variable and i an integer variable. Equivalence of queries now means that the queries yield the same answers given the same relations and values for the integer variables. Show that untyped relational calculus and untyped relational algebra are equivalent.

Exercise 18.16 Show that ex_{ij} is not redundant in the untyped algebra.

$$\underset{I}{\begin{array}{ccc}
\alpha_1 & a & \psi_1 \\
\alpha_1 & b & \psi_1 \\
\alpha_1 & b & \psi_2 \\
\alpha_1 & c & \psi_2 \\
\alpha_1 & c & \psi_3 \\
\alpha_1 & d & \psi_3 \\
\alpha_1 & d & \psi_4 \\
\alpha_1 & a & \psi_4 \\
\alpha_2 & a & \psi_5 \\
\alpha_2 & b & \psi_5 \\
\alpha_2 & b & \psi_6 \\
\alpha_2 & c & \psi_6 \\
\alpha_2 & c & \psi_7 \\
\alpha_2 & d & \psi_7 \\
\alpha_2 & d & \psi_8 \\
\alpha_2 & a & \psi_8
\end{array}}
\Longrightarrow
\underset{J}{\begin{array}{ccc}
\beta & a & \theta_1 \\
\beta & b & \theta_1 \\
\beta & b & \theta_2 \\
\beta & c & \theta_2 \\
\beta & c & \theta_3 \\
\beta & d & \theta_3 \\
\beta & d & \theta_4 \\
\beta & a & \theta_4
\end{array}}$$

Figure 18.6: Another query not expressible in *while*$_{new}$

♠ **Exercise 18.17** Sketch a proof that *while*$_{uty}$ and the language QL described in Remark 18.3.2 are equivalent.

Exercise 18.18 Write a QL program computing the transitive closure of a binary relation.

♠ **Exercise 18.19** This exercise concerns well-behaved *while*$_{uty}$ programs. Show the following:

(a) It is undecidable whether a given *while*$_{uty}$ program is well behaved.

(b) Each syntactically well-behaved *while*$_{uty}$ program is well behaved.

(c) For each well-behaved *while*$_{uty}$ program, there exists an equivalent syntactically well-behaved *while*$_{uty}$ program.

Exercise 18.20 Write a *while*$_{uty}$ program that constructs the relation $\widehat{succ}$ from $R_<$ in the proof of Theorem 18.3.3.

♠ **Exercise 18.21** [AV91b] Prove that any query on a unary relation computed by a *while*$_{new}$ or *while*$_{uty}$ program in polynomial space is in FO. (For the purpose of this exercise, define the space used in a program execution as the maximum number of occurrences of constants in some instance produced in the execution of the program.) Note that, in particular, *even* cannot be computed in polynomial space in these languages.

♠ **Exercise 18.22** [AV91a] Consider the following extension of datalog$^{\neg\neg}$ with the ability to create new values. The rules are of the same form as datalog$^{\neg\neg}$ rules, but with a different semantics than the active domain semantics used for datalog$^{\neg\neg}$. The new semantics is the following. When rules are fired, all variables that occur in heads of rules but do not occur positively in the body are assigned distinct new values, not present in the input database, program, or any of the other relations in the program. A distinct value is assigned for each applicable valuation of the variables positively bound in the body in each firing. This is similar to the *new* construct in $while_{new}$. For example, one firing of the rule

$$R(x, y, \alpha) \leftarrow P(x, y)$$

has the same effect as the $R := new(P)$ instruction in $while_{new}$. The resulting extension of datalog$^{\neg\neg}$ is denoted datalog$^{\neg\neg}_{new}$. The *well-behaved* datalog$^{\neg\neg}_{new}$ programs are those that never produce new values in the answer. Sketch a proof that well-behaved datalog$^{\neg\neg}_{new}$ programs express all queries.